Domaine Sciences et Technologies
Master Bio-informatique : DLAD

Aix*Marseille université
Socialement engagée

Programmation avancée : UML
Code UE : SBICU05

Année 2023-24

Unified Modeling Language

Introduction

L'UML (Unified Modeling Language) est une norme émanant de l'OMG (Object Management Group, voir https://www.omg.org). L'OMG, organisme à but non lucratif, a été créé en 1989 à l'initiative de grandes sociétés. Son rôle est de promouvoir des standards qui garantissent l'interopérabilité entre applications orientées objet, développées sur des réseaux hétérogènes. Actuellement, nous en sommes à la version 2.5.1 de la norme et sa spécification complète est disponible au lien suivant : https://www.omg.org/spec/UML/2.5.1/PDF. L'UML est un langage formel défini par un méta-modèle (modèle d'un langage de modélisation). Il permet en autres d'exprimer et d'élaborer des modèles objet, indépendamment de tout langage de programmation. C'est un support de communication performant qui, par ses représentations graphiques, permet de faciliter la représentation et la compréhension de solutions objets. UML, en tant que cadre méthodologique pour une analyse objet permet de représenter un système selon différentes vues complémentaires : les diagrammes.

Types de diagramme de classes

Un diagramme UML est une représentation graphique, qui s'intéresse à un aspect précis du modèle ; c'est une vue du modèle. Combinés, les différents types de diagrammes UML offrent une vue complète des aspects statiques et dynamiques d'un système. On dénombre 13 types de diagrammes différents permettant de représenter le cahier des charges du projet, les classes et la manière dont elles s'agencent et interagissent entre elles. Dans cette optique, dans ce cours, nous ne détaillerons que quelques types de diagrammes. Voici tout de même une liste de tous les diagrammes disponibles regroupés dans trois grandes familles et leur utilité.

Les diagrammes structurels ou statiques

- Les diagrammes de classes : ils représentent les classes qui vont constituer le système.
- Les diagrammes d'objets : ils permettent de représenter les instances de classes (objets) utilisées dans le système.
- Les diagrammes de composants : ils modélisent les composants d'un système dans la durée (par composant on entend composant physique ou logique).
- Les diagrammes de déploiement : ils permettent de représenter le déploiement du système sur des éléments matériels (ordinateurs, périphériques, etc.).
- Les diagrammes des paquetages : un paquetage est un conteneur logique permettant de regrouper et d'organiser les éléments dans le modèle UML. Le Diagramme de paquetage sert à représenter les dépendances entre paquetages.
- Les diagrammes de structure composite : ils décrivent la structure interne des relations entre les composants d'une classe.

Les diagrammes comportementaux

— Les diagrammes des cas d'utilisation : ils décrivent la fonctionnalité du système qu'un acteur (intervenant extérieur du système) doit exécuter pour obtenir un certain résultat.
— Les diagrammes d'états : ils permettent de décrire sous forme de machine à états finis le comportement du système ou de ses composants.
— Les diagrammes d'activité : ils permettent de décrire sous forme de flux ou d'enchaînement d'activités le comportement du système ou de ses composants.

Les diagrammes d'interaction

— Les diagrammes de séquence : ils indiquent l'interaction entre plusieurs partenaires de communication.
— Les diagrammes de communication : anciennement appelé diagramme de collaboration, il est une sorte de diagramme de séquence simplifié où l'on va se concentrer sur les échanges de messages entre objets.
— Les diagrammes d'interaction globale : ils permettent de décrire comment les scénarios identifiés dans le diagramme de séquences peuvent s'enchaîner.
— Les diagrammes de temps : il permet de décrire les variations d'une donnée au cours dutemps.

PlantUML

Afin de produire les diagrammes que nous verrons dans le cours, nous allons utiliser l'outil *PlantUML* (lien : https://plantuml.com/fr/). PlantUML est un outil *open-source* qui permet de générer des diagrammes à partir d'un fichier texte. Les diagrammes UML supportés sont les suivants :

— diagrammes de séquence
— diagrammes de cas d'utilisation
— diagrammes de classes
— diagrammes d'objet
— diagrammes d'activité
— diagrammes de composant
— diagrammes de déploiement
— diagrammes d'état
— diagrammes de temps

Nous allons nous concentrer sur les diagrammes de classes et de séquences.

Diagramme de classes

Représentation d'une classe

Le diagramme de classes permet d'exprimer la structure d'un projet en termes de classes et de relations entre ces classes et donc d'effectuer une réflexion sur l'architecture logicielle. Un diagramme de classes va pouvoir comporter tous les éléments de la programmation orienté objet. Les classes seront représentées par des rectangles composés de trois parties :

— le premier compartiment représente le nom de la classe(pour les interfaces et les classes abstraites, ce nom sera noté en italique et en plus, pour les interfaces, il sera précédé de la notation **I** ou “interface”).
— le 2ème compartiment représente les attributs de la classe
— le 3ème compartiment représente les méthodes de la classe

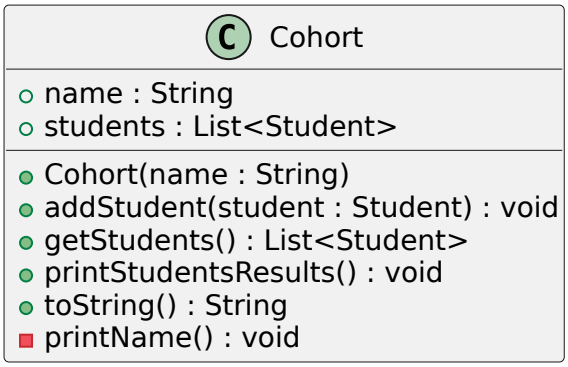
La visibilité des attributs et des méthodes est indiquée à l'aide d'un caractère précédent le nom de l'attribut ou de la méthode :

— **public** (mot-clé `public`), symbole ∘ ou `+` : l'élément est visible depuis toutes les classes.
— **protégé** (mot-clé `protected`), symbole ◇ ou `#` : l'élément est visible pour les sous-classes de la classe et les classes du même *package*.
— **paquetage privé** (mot-clé aucun : visibilité par défaut lorsque aucune visibilité n'est indiquée), symbole △ ou `~` : l'élément est visible pour les classes du même *package*.
— **privé** (mot-clé `private`), symbole ◇ ou `-` : l'élément n'est visible que dans la classe dans laquelle il est défini.

La syntaxe pour les éléments d'une classe est la suivante :

— **Syntaxe pour les attributs d'une classe** (entre accolades, les mentions optionnelles) :
<visibilité> <nom_attribut> : <type> [`[`<multiplicité>`]`] [= <valeur_par_défaut>]
On peut représenter les attributs de classe (*static* en Java) en soulignant la déclaration de l'attribut.
Exemples :
— ‘- i : int‘ pour un attribut privé
— ‘+ point : Point2D‘ pour un attribut public
— `integers : int[*]` pour un tableau d'entiers de taille quelconque et à la visibilité non définie
— `# points : Point2D[4]` pour un tableau de 4 points exactement et à la visibilité protégée
— `~ x : float = 0.0` pour un nombre à virgule initialisé à 0 et à la visibilité paquetage privé
— **Syntaxe pour les méthodes ou constructeurs d'une classe** (entre accolades, les mentions optionnelles) :
<visibilité> <nom_méthode> ([<paramètre_1>, . . . , <paramètre_N>]) : [<type_renvoyé>] [{<propriétés>}]
La syntaxe de définition d'un paramètre (<paramètre>) est la suivante :
[<direction>] <nom_paramètre> : <type> [‘[’<multiplicité>’]’] [=<valeur_par_défaut>]
On peut représenter les méthodes de classe (`static` en Java) en soulignant la déclaration de la méthode.
Pour une méthode abstraite (`abstract` en Java), on met la déclaration de la méthode en *italique*.
Exemples :
— ‘+ toString() : Stringpour une méthode publique, sans paramétre et retournant une chaîne de caractéres de typeString‘
— `~ setAbscisse(x : float)` pour une méthode visible dans le paquetage, et prenant un nombre à virgule de type `float` en paramètre

La classe `Cohort` de la première planche de TP est représenté par le diagramme ci-dessous (généré à l'aide de PlantUML).



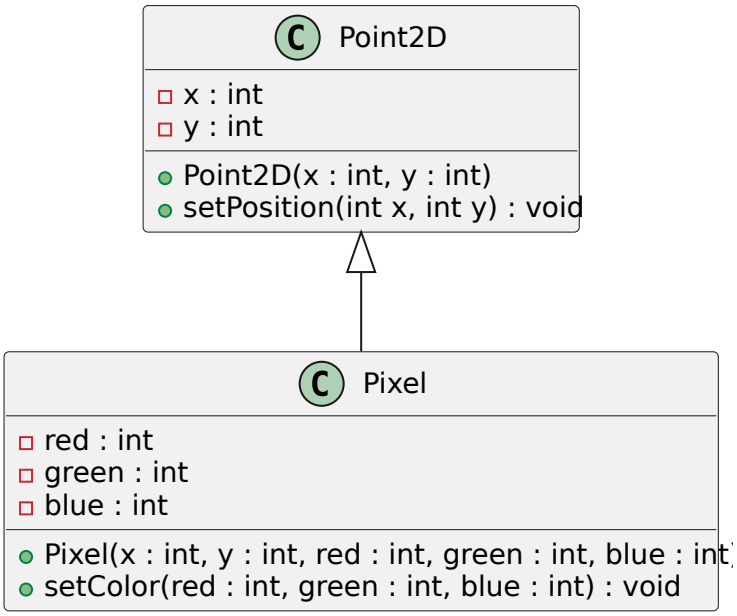
Un des apports les plus importants des diagrammes de classe est de décrire la relation entre les classes d'un projet de développement. Il existe de nombreux types de relations que peuvent avoir les classes.

Relation d'extension

L'extension décrit une relation entre une classe parent et une classe enfant. La classe enfant est intégralement cohérente avec la classe parent, mais comporte des informations supplémentaires (attributs, méthodes, ...). Une classe enfant peut redéfinir (même signature) une ou plusieurs méthodes de la classe parent. Sauf indication contraire, un objet utilise les méthodes les plus spécialisées dans la hiérarchie des classes. Toutes les associations de la classe parent s'appliquent aux classes enfants. Un objet de la sous-classe peut être utilisé partout où un objet de la classe de base est autorisé (principe que l'on verra plus en détails dans la suite du cours).

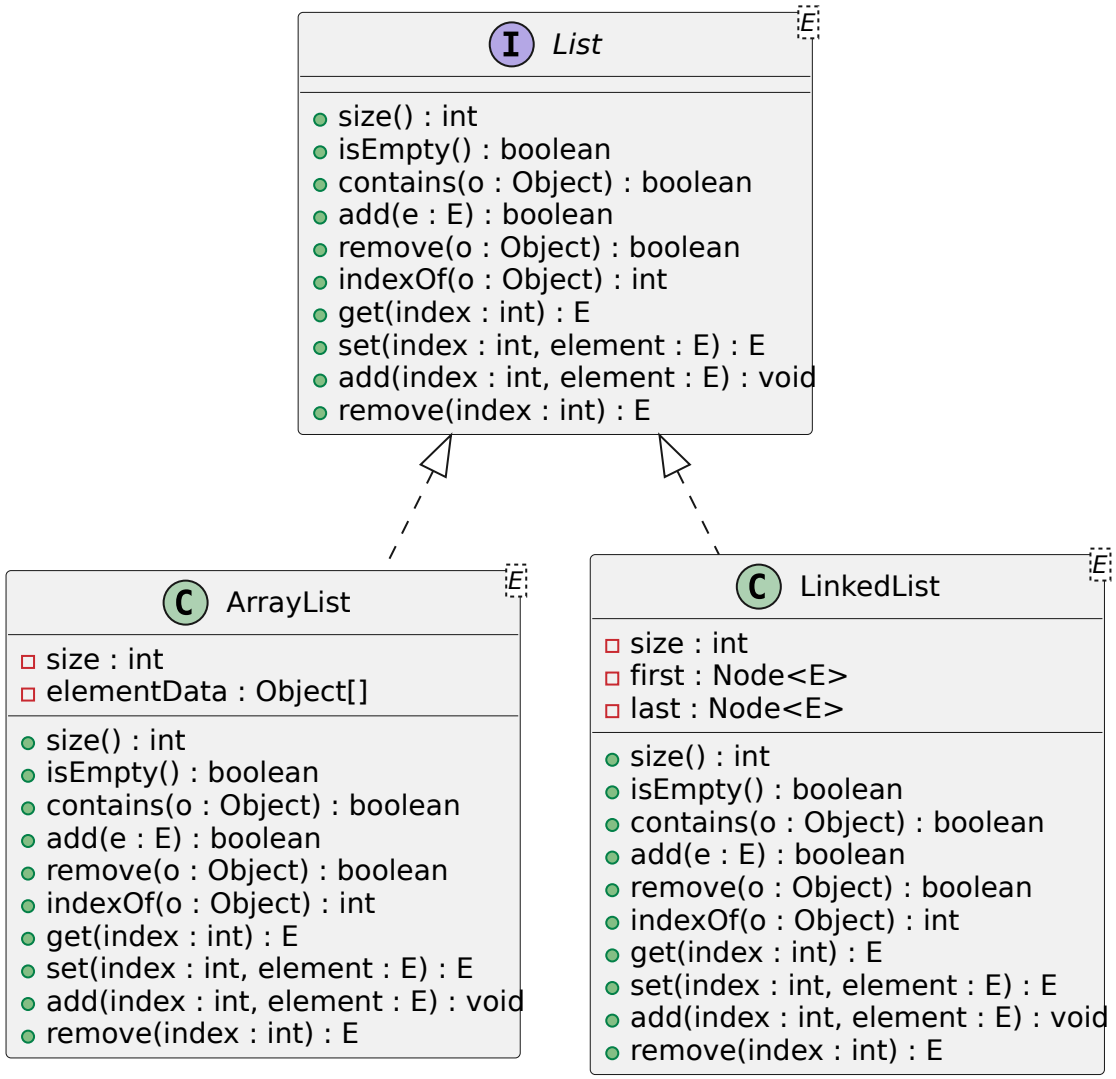
Le symbole utilisé pour la relation d'extension est une flèche avec un trait plein dont la pointe est un triangle vide (blanc) vers la classe parent.

Le diagramme ci-dessous donne la relation entre une classe parent permettant de représenter un point du plan (`Point2D`) et une classe enfant permettant de représenter un pixel (`Pixel`).



Relation d'implémentation

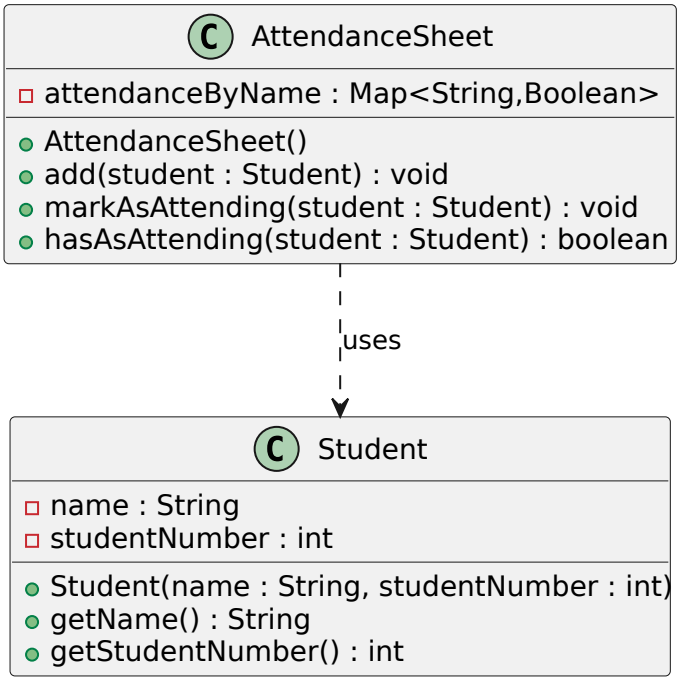
Une relation d'implémentation existe entre deux classes lorsqu'une d'entre elles doit implémenter ou réaliser le comportement spécifié par l'autre. Dans le cas de Java, c'est généralement la relation entre une classe et une des interfaces qu'elle implémente. Une relation d'implémentation est représentée par une flèche composée d'un trait composé de tirets et d'un triangle vide. Le connecteur part du client (qui réalise le comportement) vers le fournisseur (qui spécifie le comportement). Dans le diagramme ci-dessous, nous avons représenté les classes `ArrayList` et `LinkedList` qui implémentent toutes deux l'interface `List` (toutes les méthodes de `List` ne sont pas décrites afin d'éviter de rendre trop complexe le diagramme).



Relation de dépendance

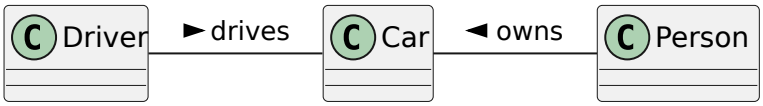
Une dépendance est une relation indiquant qu'une classe en utilise une autre classe comme type dans les arguments d'une de ses méthodes. L'exemple ci-dessous montre la relation entre une classe `AttendanceSheet` permettant de représenter des feuilles de présence et une classe `Student` permettant de représenter des étudiants. Le symbole utilisé pour la relation de dépendance est une flèche avec un trait en tirets partant de la classe dépendante. On ajoute généralement du texte (dans notre exemple *uses*) afin de préciser la nature exacte de la

relation entre les classes.

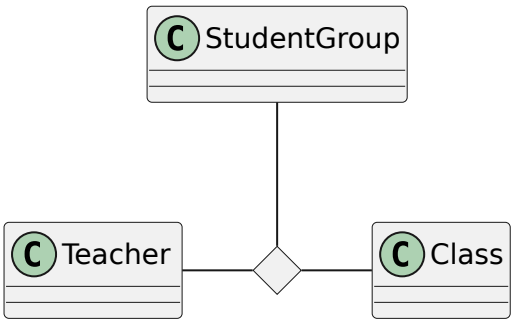


Relation d'association

Une association est une relation structurelle entre objets. Ce type de relation peut être binaire, dans ce cas, elle est représentée par un simple trait, ou n-aire, auquel cas les classes sont reliées à un losange par des traits simples. La relation peut être nommée et on peut indiquer le sens de lecture de son nom avec un symbole ▶. Par exemple, le diagramme ci-dessous indique les relations entre des voitures, des conducteurs et des propriétaires de voitures à l'aide d'associations binaires.

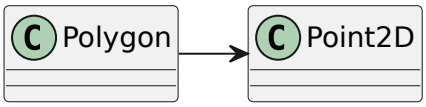


Le diagramme ci-dessous illustre une relation ternaire entre un cours, l'enseignant assurant le cours et les étudiants assistant au cours.



Il est possible d'indiquer un sens à la relation. Dans une relation binaire, on peut indiquer à l'aide d'une flèche que seuls les objets d'une classe source a la connaissance des objets de la classe destination. Dans l'exemple

ci-dessous, seuls les objets de classe `Polygon` connaissent ceux de la classe `Point2D` auxquels ils sont liés, mais pas l'inverse.



Relation d'agrégation

L'agrégation est une relation d'association dans laquelle une classe `Part` est contenue dans une classe `Aggregate`. Plus précisément, on a les propriétés suivantes :

- une (ou plus) instance(s) de `Part` est stockée dans une instance de `Aggregate` ;
- Chaque instance de `Part` peut appartenir à plusieurs instances de `Aggregate` ;
- La durée de vie des instances de `Part` n'est pas gérée par les instances de `Aggregate` qui l'a contienne (les instances `Part` contenues dans un objet `Aggregate` ne sont pas détruites quand celui-ci est détruit) ;
- Les instances de `Part` n'ont pas connaissance de l'existence des instances de `Aggregate`.

Pour résumer, lorsqu'il y a une relation d'agrégation, une instance d'une classe contient des objets d'une autre classe, mais ceux-ci ont leur vie propre. Comme exemple intuitif pour une agrégation, on peut considérer des blocs de construction d'une maison jouet. La maison jouet est construite à partir de blocs. Il est possible de la démonter entièrement, mais les blocs resteront disponibles pour une autre construction.

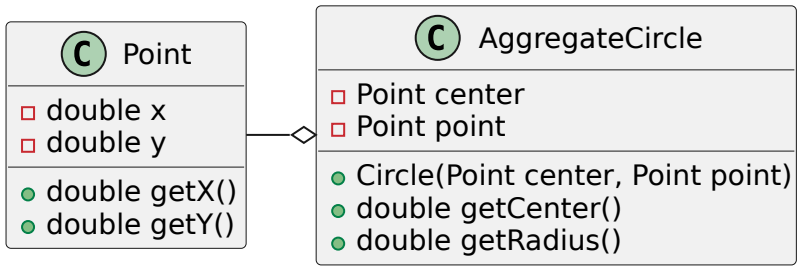
On représente cette relation par un trait reliant la classe `Aggregate` à la classe `Part` avec un losange vide à l'extrémité connecté à `Part`.

Pour un exemple plus proche de la programmation orientée objet, on peut considérer les classes `Point` et `AggregateCircle` suivantes :

```
1  public class Point {
2    public final double x, y;
3
4    public Point(double x, double y){
5      this.x = x;
6      this.y = y;
7    }
8
9    public double distanceTo(Point p){
10     double dx = this.x - p.x;
11     double dy = this.y - p.y;
12     return Math.hypot(dx, dy);
13   }
14
15   public double getX(){
16     return x;
17   }
18
19   public double getY(){
20     return y;
21   }
22 }
```

```java

public class Circle {
  private Point center, point;

  public Circle(Point center, Point point){
    this.center = center;
    this.point = point;
  }

  public Point getCenter(){
    return center;
  }

  public double getRadius(){
    double dx = center.x - point.x;
    double dy = center.y - point.y;
    return Math.hypot(dx, dy);
  }
}
```

Le diagramme de classes correspondant est le suivant :



Relation de composition Tout comme l'agrégation, la composition est une relation d'association dans laquelle une classe `Part` est contenue dans une classe `Composite`. La différence est que pour la composition le cycle de vie des parties est dépendant du cycle de vie des objets composés. Plus précisément, on a les propriétés suivantes :

- une (ou plus) instance(s) de `Part` est stockée dans une instance de `Composite` ;
- Chaque instance de `Part` appartient à une seule instance de `Composite` ;
- Chaque instance de `Part` a sa durée de vie (construction et destruction) gérée par l'instance de `Composite` qui l'a contienne;
- Les instances de `Part` n'ont pas connaissance de l'existence des instances de `Composite`.

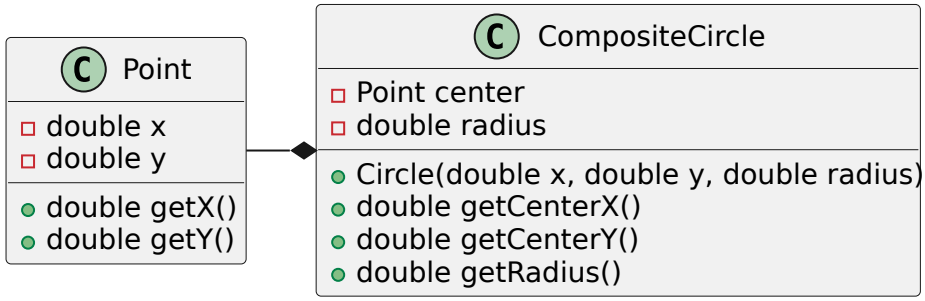
Pour résumer, lorsqu'il y a une relation de composition, une instance d'une classe contient des objets d'une autre classe, mais ceux-ci ont leur vie propre. Comme exemple intuitif pour une composition, on peut considérer une maison contenant une ou plusieurs pièces. La durée de vie d'une pièce est contrôlée par la maison, car elle ne peut pas exister sans la maison.

On représente cette relation par un trait reliant la classe `Aggregate` à la classe `Part` avec un losange vide à l'extrémité connecté à `Part`.

Pour un exemple plus proche de la programmation orientée objet, on peut considérer la classe `Point` déjà vue pour l'agrégation avec la classe `CompositeCircle` suivante :

```java
public class CompositeCircle {
  private Point center;
  private double radius;

  public Circle(double x, double y, double radius){
    this.center = new Point(x, y);
    this.radius = radius;
  }

  public double getCenterX(){
    return center.getX();
  }

  public double getCenterY(){
    return center.getY();
  }

  public double getRadius(){
    return radius;
  }
}
```

Le diagramme de classes correspondant est le suivant :



Multiplicités

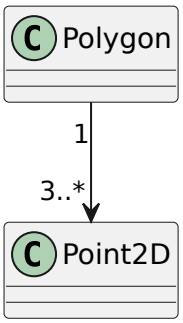
Lorsqu'on définit une relation d'association (qui peut être une agrégation ou une composition), il est souvent utile de définir la multiplicité, c'est-à-dire le nombre d'objets possibles pour chaque extrémité de la relation. Dans une association binaire, la multiplicité sur une extrémité contraint le nombre d'objets de la classe connecté à l'extrémité pouvant être associés à un seul objet de l'autre classe. La syntaxe pour une multiplicité est `min..max`, où `min` et `max` sont des nombres représentant respectivement les nombre minimaux et maximaux d'objets concernés par l'association. On peut utiliser une `*` à la place de `max` pour signifier que le nombre d'objets n'est pas borné. De plus, `n..n` se note aussi `n` et `0..*` se note aussi `*`.

Exemples de multiplicités :

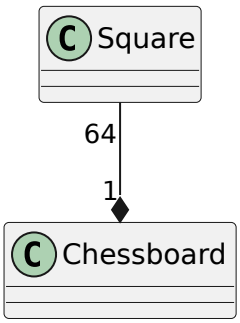
— exactement un : `1` ou `1..1` ;
— plusieurs : `*` ou `0..*`;

— au moins un : `1..*` ;
— de un à six : `1..6`.

Dans l'exemple ci-dessous, on donne les multiplicités de la relation entre `Point` et `Polygon` en précisant qu'un point n'est que dans un polygone et qu'un polygone contient au moins 3 points.



Dans l'exemple ci-dessous, on donne les multiplicités de la relation entre un échiquier (classe `Chessboard`) et une case (classe `Square`).



Exemple de diagramme de classe

Pour cet exemple, nous allons considérer le problème suivant de commande d'articles en ligne. Il est défini avec les contraintes suivantes :

— Un catalogue regroupe des articles, il permet de trouver un article à partir de sa référence.
— Un article est caractérisé par un prix et une référence que l'on peut obtenir. On veut aussi pouvoir déterminer si un article est plus cher qu'un autre.
— Une commande est créée pour un client et un catalogue donnés, on peut ajouter des articles à une commande, accéder à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.
— Un client peut créer une commande pour un catalogue et commander dans cette commande des articles à partir de leurs références.

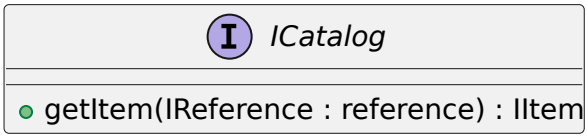
Pour réaliser le diagramme de classes, on va tout d'abord déterminer quelles sont les interfaces dont on va avoir besoin pour satisfaire aux besoins posés. La première question à se poser est de déterminer quels sont les objets nécessaires à la résolution du problème. Pour ce problème, on peut identifier les objets suivants : catalogue, article, commande, référence, client, prix. On peut donc considérer 5 types d'objets différents et par conséquent

5 interfaces : `ICatalog`, `IItem`, `IOrder`, `IReference`, `IPrice`, `IClient` (on met souvent un `I` devant le nom des interfaces afin de les différencier des classes concrètes).

La deuxième question à laquelle on doit répondre est de déterminer quelles sont les méthodes des interfaces. Pour cela, il suffit généralement d'extraire de la définition du problème les actions (donc les verbes) que l'on peut réaliser sur les objets.

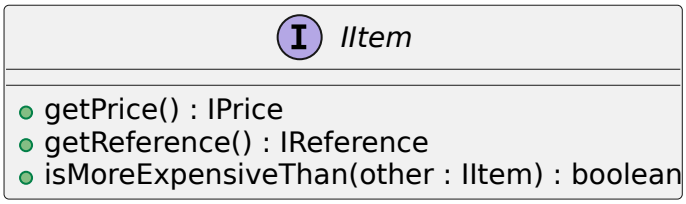
Un catalogue regroupe des articles, il permet de **trouver** un article à partir de sa référence.

⇒ L'interface pour les catalogues peut donc être la suivante :



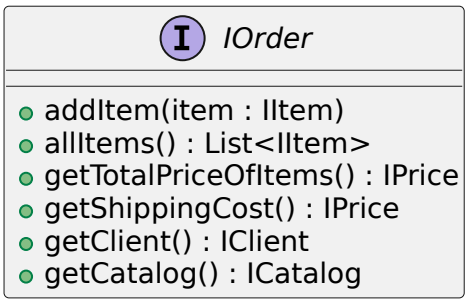
Un article est caractérisé par un prix et une référence que l'on **peut obtenir**. On veut aussi pouvoir **déterminer** si un article est plus cher qu'un autre

⇒ L'interface pour les articles peut donc être la suivante :



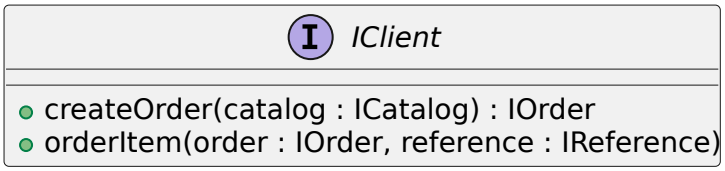
Une commande est **créée** pour un client et un catalogue donnés, on peut **ajouter** des articles à une commande, **accéder** à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.

⇒ L'interface pour les commandes peut donc être la suivante :

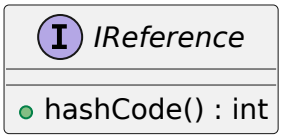


Un client peut **créer** une commande pour un catalogue et **commander** dans cette commande des articles à partir de leurs références.

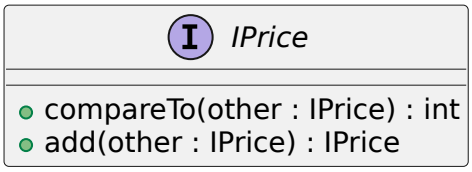
⇒ L'interface pour les clients peut donc être la suivante :



Pour les références, il y a moins de précisions dans la description du problème. Puisqu'une référence sert de clé dans un catalogue, il semble logique qu'elle implémente la méthode `hashCode` permettant l'utilisation de l'objet dans une table de hashage.

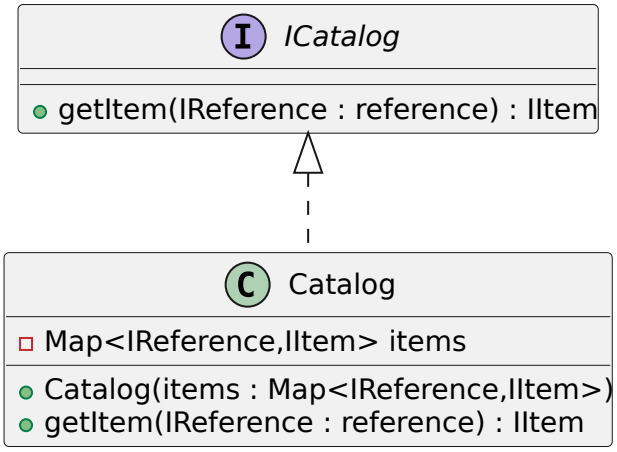


Pour les prix, les deux opérations qui semblent essentielles sont la possibilité de comparer et d'ajouter des prix.

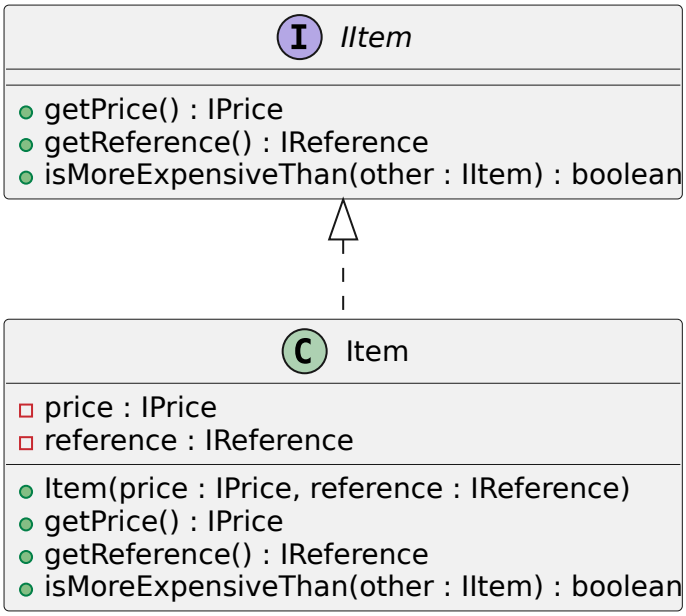


Maintenant qu'on a défini des interfaces, il faut définir les classes qui vont les implémenter en choisissant la structure de leurs données (leurs attributs).

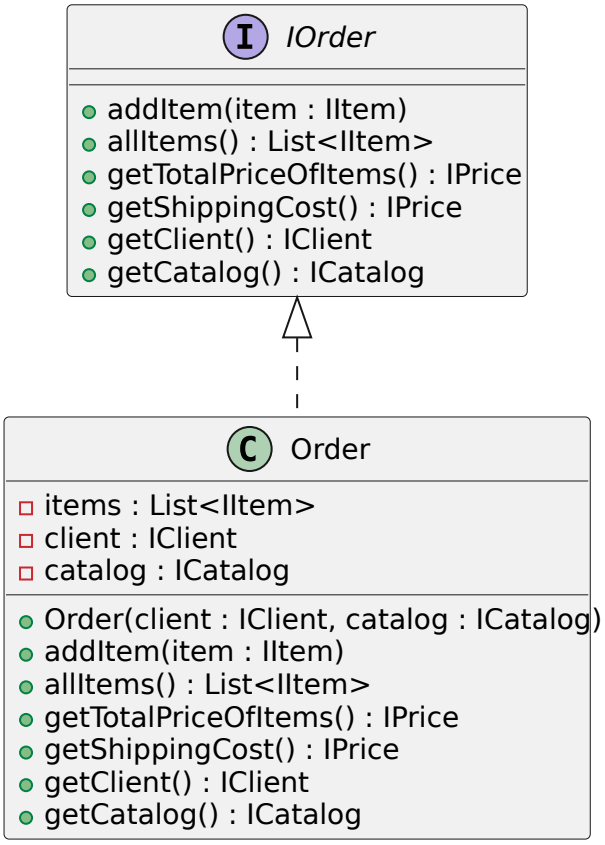
— Classe `Catalog`



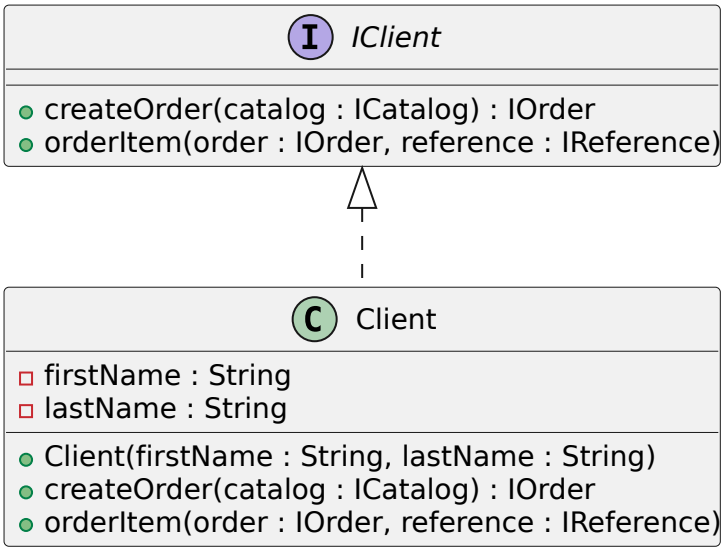
— Classe `Item`



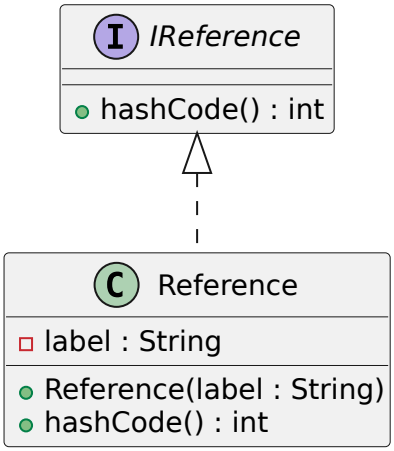
— Classe `Order`



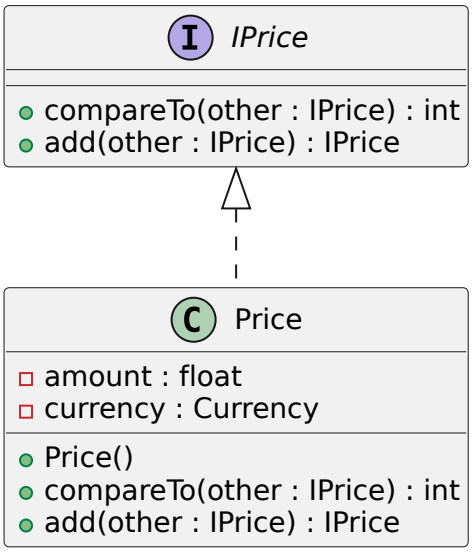
— Classe `Client`



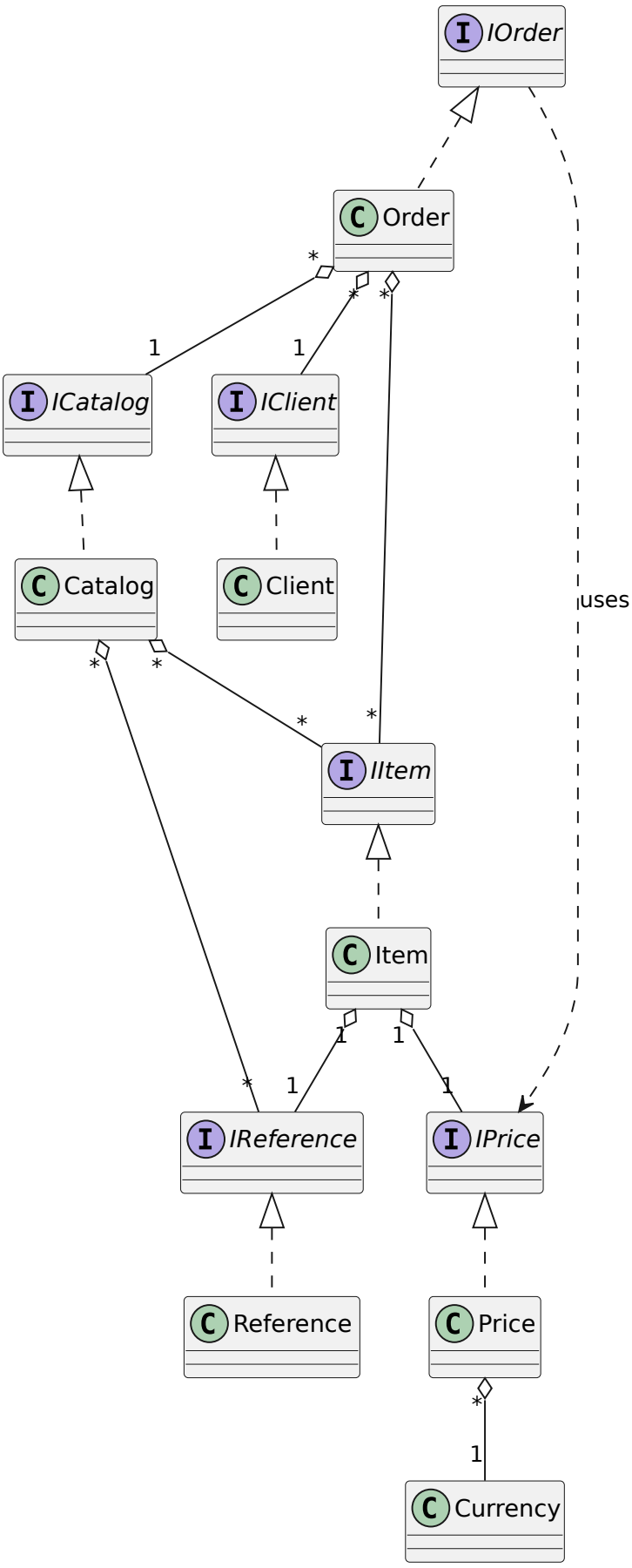
— Classe `Reference`



— Classe `Price` (on utilise la classe `Currency` de Java).



En ajoutant les relations avec leur multiplicité, on obtient le diagramme suivant :


IOrder
Order
ICatalog
IClient
Catalog
Client
IItem
Item
IReference
IPrice
Reference
Price
Currency
uses
1
*